Sorcery Design Document

Tommy Su, Rui Li, Jason Pan

CS246 - F23

December 5, 2023

1. Overview

1.1 General Structure

The high-level overview of our program remained the same as before, where we applied the MVC (Model View Controller) design architecture for creating our program.

The Model is the Player class, which stores the current game state of each player. This includes their health and magic values, and the contents of the Card pointer vectors that denote which cards are on the board, in each of the zones: the board (**note:** the board is called “field” in our code), graveyard, hand, ritual, and deck.

The View in our program consists of the TextDisplay and GraphicsDisplay classes, both of which are subclasses of the Observer superclass. They are responsible for displaying the current game state for both players as output to the user.

The Controller in our program is the GameManager class, which processes input from the user and updates the Player objects (Model) accordingly. The Player class contains vectors of Card pointers that then notify the TextDisplay and GraphicDisplay (View) of changes to the game state (e.g., new minions are summoned on the board). The View then updates itself, and then outputs the new version of the game state to the user.

1.2 Module Descriptions

1.2.1 GameManager

The GameManager class is the Controller in our program; it processes commands from the user, and calls the Player to perform tasks based on input. This class owns a TextDisplay object, a GraphicsDisplay object, and two Player objects, and is also responsible for initialising these objects. This class also contains a vector of CardObservers, and tells the players to use the Cards’ triggered abilities when a game state or condition is reached (e.g., a turn ends).

1.2.2 Player

The Player class is the Model in our program; it manages the states of Cards in different zones (board, hand, graveyard, ritual, or deck) of a particular player. These zones are represented using vectors of Card pointers. The Player object calls functions like play(), use(), useAbility(), attack(), and drawCard() to modify the contents of each zone. These functions are overloaded with different parameters for cases (e.g., play a card with/without a target). The Player class also manages a player’s health and magic values.

1.2.3 Observer

The Observer is a superclass of TextDisplay and GraphicsDisplay, so they can update their own game states to be outputted to the user.

1.2.4 CardObserver

The CardObserver is a superclass of the Card class for Cards to observe game board conditions to cast triggered abilities.

1.2.5 TextDisplay

The TextDisplay class is one interface of View in our MCV model, responsible for outputting ASCII art that reflects the current game state in the terminal. It is a subclass of the Observer superclass. This class attaches itself to the Cards that are active on the board, ritual, and graveyard, and is notified when there is a change to game state (e.g., when a Minion dies and is sent to the Graveyard), so its state can be updated accordingly. It is also attached to the Player objects to observe health and magic changes.

1.2.6 GraphicsDisplay

The GraphicsDisplay class is the other interface of the View in our MCV model, similar to TextDisplay. It outputs the current game state using xWindow. It is a subclass of the Observer class, and attaches itself to the cards that are active on the board, ritual, and graveyard. It is notified when the game state changes. It is also attached to the Player object to observe health and magic changes.

1.2.7 Library

The Library is responsible for managing all the unique Cards that are available in the game. It contains four std::maps, one for each of the four Card subclasses: Minion, Spell, Enchantment, Ritual. Each unique Card is initialised in the Library. When Players are initialised in the GameManager, their decks are built by invoking Card copy constructors to create a copy of the Cards in heap memory, and pointers to these heap-allocated cards are added to a Player’s deck.

1.2.8 Card

The Card class has 4 subclasses: Minion, Spell, Enchantment, Ritual. Card owns-an Ability. Card objects notify the TextDisplay and GraphicsDisplay to update when their game state changes (e.g., when the Card moves to another zone). The Card class also observes the GameManager so that they can cast abilities that require a specific game stage or condition (e.g., when a player’s turn starts).

1.2.9 Minion, Spell, Enchantment, Ritual

These classes are the 4 subclasses that inherit fields and methods from Card, overriding methods as necessary (e.g., Minions override the modifyAttack() method). The functionalities of all these Card subclasses are abstracted into an ability class that determines the effect the Card has (e.g., statModifyEffect will handle all changes to a Minion’s attack and defence values).

1.2.10 Ability
The Ability class owns-an effect subtype. Abilities have a variant field, EffectHolder, that allows it to hold different classes of effects (e.g., StatModifyEffect, MoveEffect). Abilities are defined by fields like Target (which determines what Cards to apply the effect on), AbilityType (which determines if it's a triggered or active ability), and Condition (which determines what causes the ability to be casted, such as onTurnBegin). When the Player object calls a use or play function of a Card, the player then calls a useAbility() function depending on the card's specifications to use the Ability's effect.

1.2.11 Effect
There are various abstracted effect types (these effects are their own classes), that perform specific jobs. An example is the AbilityCostModifyEffect, which will increase or decrease a Card's ability cost.

In particular, there is a special RemoveEnchantEffect that checks which enchantment has been applied on the Minion last, and performs the reverse effect of the Enchantment (e.g., will subtract both attack and defence by 2 if Giant Strength was the Enchantment that was used). This will be used for when Enchantments are removed from Minions.

1.3 Differences in Final Design
There were a few major changes that we made to our program design.

First, we removed unnecessary classes from our program. The first was the CardList class — it was essentially just a vector of Card pointers with no additional functionality. Originally, we created the class to add Observers that would notify each CardList to update its vector of Card pointers to keep track of the state of each zone. However, we realised this design pattern was obsolete since we could just move pointers between different zones using the moveCard() function in the GameManager class; we didn't need Observers for this. Next, we removed the "Board" class in our original design, which we defined as a class containing all of the zones of a Player. We thought this class was unnecessary, as we could just put all of the Card pointer vectors that denoted the state of different zones in their respective Player object instead.

Next, we implemented a Library, which contained a std::map for every card type (minion, spell, enchantment, ritual) which contained an entry for each unique card that we needed to implement. We decided to do this because we realised that within each Card subclass, the individual Cards shared similar fields and methods; only the values of the fields changed. As a result, implementing each individual card as a separate subclass served no purpose in our design.

One final major change was the way we applied abilities. Originally, we defined a class of Target objects that we put inside Ability objects (via aggregation). We wanted to use the Target class to determine which Cards to apply abilities on, but we found that it was inefficient, as we needed to pass a pointer to the zones the Cards were located in. Instead, we decided to implement a new targeting system. Recall that Cards have an Ability field, and Ability objects contain an effect. When a Player object uses the play() function with a Card, the Card's definition determines which useAbility() function needs to be called, based on how many targets are affected (e.g., Blizzard targets all minions that are played, while Enrage targets just one minion). After useAbility() is called, the effect is applied.

1.4 Low Coupling and High Cohesion

To guarantee low coupling, we used a variety of techniques, one of which includes abstraction. An example of this is the Ability class, where the Player class just needs to call a useAbility() function to use the effect. Hence, the Player isn't likely to be affected by changes to the specific implementation of the Ability class, ensuring low coupling. We also used encapsulation by making class fields private, and defining getter and setter functions in each class. This ensures the number of dependencies for each module is limited, contributing to low coupling.

To ensure high cohesion in our program, we applied the Single Responsibility Principle (SRP), where each module has a single responsibility to change. For instance, the GameManger handles input from the user, the Player class manages and keeps track of a player's game state, and the Library is used for defining individual cards in each Card subclass (Minion, Spell, Enchantment, Ritual). Each of these modules perform a different task, contributing to high cohesion between the different classes.

2. Design

In our project, we utilised the Observer design pattern to solve the issue of handling multiple interfaces. The TextDisplay and GraphicDisplay objects, both of which are Observers, are created in the GameManager. They are attached to Player objects to observe changes in player health and magic, and are also attached to Cards played on the board, graveyard, hand, and ritual zones to observe changes to the state of Cards in each zone.

We also used the Observer pattern to implement the triggered abilities of Minions and Rituals. When placed on the board, these Card objects will be attached to the GameManager to observe changes to the game state that trigger its specific ability. When a game state occurs, such as a new turn begins, the GameManager then notifies all CardObservers that are subscribed to the onTurnBegin event. The Cards will then call its triggered ability and apply its effects accordingly.

Next, we used the Template design pattern to organise and implement each subclass of Cards. The four subclasses of the Card class, Minion, Spell, Enchantment, and Ritual, all override methods of the Card class during runtime. For example, a Minion overrides the modifyAttack() and modifyDefense() functions to modify its stats when a Card like Enrage is applied on it. We decided to implement Cards in this manner because Card subtypes all shared similar features: they had a magicCost, and they were all able to carry Abilities.

Next, we ran into an issue where it was inefficient to define unique Cards as their own subclasses. We noticed that we didn't define any new methods or fields for these unique Cards, since their immediate superclass (Minion, Spell, Enchantment, Ritual) already contained the necessary fields for that Card (e.g., attack and defence values for Minion Cards). To mediate this, we created a Library that contained four std::maps for each Card subclass. In conjunction with abstracting abilities and effects, this design made it easy for us to add new unique Cards to the libraries, as we just needed to add a new entry in the respective std::map, define the Cards by their subtype, and define their Abilities and Effects. As Cards have more specific functionalities, we would just need to create new Effect types that reflected the functionality of the Cards.

3. Resilience to Change

One aspect of our design that supports changes to program specification is its scalable nature. Since we created a std::map in the Library for each of different types of cards, abstracted all abilities of Minions, Spells, Enchantments, and Rituals, and defined effects by their core functionalities, it made it easy to add new cards to the game.

This is especially effective if Cards share similar behaviour. For example, Giant Strength and Enrage both modify a minion's health, so we use the same StatModifyEffect to handle the functionalities of both cards. By using abstractions, and defining Cards by their base functionality, it makes our program easy to scale and adjust for different specifications.

Another aspect of our design that supports changes to program specification is our robust error handling. We made use of try/catch blocks and exception throws throughout our program to catch edge cases, invalid inputs, and unspecified behaviour. By having an error checking system built in place, we can then modify error handling to tailor the program to new program specifications.

4. Answers to Questions

4.1 How could you design activated abilities in your code to maximise code reuse?

To maximise code reuse, we abstracted the Ability class so that each of the four Card subtypes' functionalities were constructed using the same Ability and Effect classes. We noticed that Minion, Spell, Enchantment, and Ritual functionalities all have some amount of magic cost, and they applied some kind of effect to another Card or zone (e.g., some Enchantment increased Minion stats). Thus, it made sense to abstract these functionalities and define them based on their trigger condition, target, and abilityType, so Abilities can be used for any Card. Passive Abilities can also be implemented in the same manner. An example of this abstraction is that many Cards shared similar functionalities like modifying the attack and defence of a Minion. Thus, we created the statModifyEffect for Abilities that adjusted a Minion's stats, like Giant's Strength and Enrage.

4.2 What design pattern would be ideal for implementing enchantments? Why?

The decorator pattern could be effective for implementing Enchantments as it would be easy to stack Enchantments on Minions by defining each individual Enchantment as Concrete Decorators. However, when we were implementing different parts of the program like the inspect hand feature, we discovered that we still needed to keep track of a vector of the Enchantments that were already applied on a Minion, hence Decorators would be redundant. Thus, we found that it was easier to just apply the effect of an Enchantment on a Minion as soon as the Enchantment is called, and keep track of the Enchantments applied using a vector on the Minion. We then reversed the effect of the Enchantments when the Enchantment was removed from the Minion.

4.3 Suppose we found a solution to the space limitations of the current user interface and wanted to allow minions to have any number and combination of activated and triggered abilities. What design patterns might help us achieve this while maximising code reuse?

Template design patterns can be used to implement Minions with any number and combination of activated and triggered abilities. Our program is well-equipped for Minions to have any number and combination of triggered and activated Abilities since we defined each type of effect as a separate class to allow Ability objects to have a vector of effect variants. We would then just need to add an Ability vector field to Minions, and create some function to select which specific Ability in the vector to use. We did not implement the Abilities this way because the input commands are unspecifed for multiple Abilities.

4.4 How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

An Observer design pattern would be useful. We actually used this design pattern for the implementation of our TextDisplay and GraphicsDisplay objects, where these two objects are Observer subclasses. We attached the TextDisplay and GraphicsDisplay as Observers onto the Player and Card objects, and they will update the state of both players and all the cards accordingly. Since we've already defined an Observer class, we would just make the new interface a new subclass of the Observer class, and attach them to the Player and Card objects, making it easy to support multiple interfaces simultaneously.

5. Extra Credit Features

For extra credit, we decided to add additional unique cards with different functionalities. The challenging part was figuring out a system that made it 1) quick for us to add new Cards, 2) ensured code reusability with existing Cards that had similar functionalities. We were able to achieve this by creating a library, which was an std::map of Cards for each of the four Card subclasses. Coupled with our abstraction of Abilities and effects, it made it easy for us to add new Cards regardless of if it was a Minion, Spell, Enchantment, or Ritual.

6. Final Questions

6.1 What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

One lesson we learned is that communication in a development team is critical. We faced an issue at the beginning of this project where each member had a different idea about the implementation of a program despite having the UML. As a result, it took extra discussion to ensure that each member was clear about the design of the program. However, something that worked well for our team was that we constantly posted updates of our features, describing what we coded, what we needed to code next, and what features depended on another member's work. This way, all members were aware of the progress of our project.

Another lesson that this project taught us is that refactoring code is very normal. More often than not, our initial ideas of designing and implementing a feature was not as efficient as we thought or was entirely incorrect. As a result, it's critical to constantly evaluate the quality of our code and find ways to make the code more clean and readable. By refactoring our code often, it made it easier for us to debug.

6.2 What would you have done differently if you had the chance to start over?

If we could start over, we would plan out our program in more detail to get a better understanding of how each function in the program should work. It would've been nice to write pseudocode for different classes to see if our ideas were actually feasible. When we started coding, we realised that a lot of the classes we planned originally were unnecessary. For example, the Board, CardList, and Target classes provided no value. Creating a more detailed UML, and figuring out *exactly* how each part functioned would've been more efficient, since we modified our design several times before we reached our final version.

Another thing we would've done differently would be to Implement error catching features at the beginning, using try/catch blocks and throw statements. In the early stages of development, we had to do a lot of repetitive line-by-line output testing to figure out which portions of code were causing the error. Adding these error catching features also forced us to think about all input cases, making it easier to detect unspecified behaviour, such as users inputting excess parameters for user commands. Having this system in place made it easier to debug our program in the later stages of development when our codebase increased in size.